Networked Applications NWEN 243

Lab Exercise 3 (HAND IN, 6pts or 6% final grade) – A TCP Server

Objectives

- Experience using TCP
- Achieve the completion of a TCP server

Requirements

- This lab an individual lab written in C.
- We will be writing programs that you execute from the shell command line.
- Your server will need to handle multiple simultaneous connections. In C we will use fork() to create a new process to service each connection. You will be supplied the code for this in the skeletons.
- You must demonstrate your work to your lab demonstrator AND submit your program.

Preliminaries

- TCP – or the transmission control protocol, is a reliable byte ordered transport layer service. Delivery and order are guaranteed.

- To use TCP you will need to use the Socket API. The socket API is modeled on the file system, so uses READ and WRITE to access the network.

- You should use your client from Lab 2 to connect to, and to test your server. You can run multiple clients, with loops to ensure your sever is properly working.

The Exercise

Your task is to complete a TCP server program that will

1. Wait for a client to connect.
    a. Fork a worker process
        i. Service the client using the connection socket.
        ii. Close the connection socket and exit process (status)
    b. Loop back around for the next connection.

What will your server do? You can duplicate the SHOUTING server if you wish, but you may also implement some other service if you wish – such as serving a file (a little dangerous) or reversing the client's string, or anything else simple – this is not the focus of the exercise.

Resources

- There is a C skeleton, in the usual place on the course lab pages.
- Look at the lecture notes, they include diagrams detailing the steps a TCP server takes. Use this information to guide your program.

Run your Server

Use:

%>./server portnum

Note that we take the port number as a command line argument, this is because you need to specify it for the client – you could use a random one, but it is easier when testing with your client to use a fixed port number. Make sure you check the port is not already used, and kill old servers that you have left sitting around.

Hand in

You should demo your working program to your lab tutor, you can demo in week 5 or 6 in your lab session. This will be worth ½ of the marks. The other ½ marks will be for your FULLY COMMENTED submitted code. You should add comments that make it clear you understand what is going on. No lab report is needed.

Note:

1. Your code submission will be due 1 week after this lab (in week 6).

Useful stuff for C

int socket(int domain, int type, int protocol);

> socket() creates an endpoint for communication and returns a socket descriptor. The domain parameter specifies a communication domain; this selects the protocol family which will be used for communication. You should use AF_INET. This is the same for a server and a client.

*int bind(int sockfd, const struct sockaddr *my_addr, socklen_t addrlen);*

> bind() gives the socket sockfd the local address my_addr. my_addr is addrlen bytes long. Traditionally, this is called "assigning a name to a socket." When a socket is created with socket, it exists in a name space (address family) but has no name assigned. It is necessary to assign a local address using bind() before a SOCK_STREAM socket may receive connections

int listen(int sockfd, int backlog);

To accept connections you first need to tell the socket to ‘listen’. The backlog parameter defines the maximum length the queue of pending connections may grow to – set this to 5.

*int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);*

The accept() system call waits until there is a connection request, and then extracts the first connection request on the queue of pending connections, creates a new connected socket, and returns a new file descriptor referring to that socket. The original socket sockfd is unaffected by this call.

*int read(int sockfd, void *buf, int count);*

read() attempts to read up to count bytes from file descriptor fd into the buffer starting at buf.

*int write(int sockfd, const void *buf, int count);*

write() writes up to count bytes to the file referenced by the file descriptor fd from the buffer starting at buf.

int close(int fd);

close() closes a file descriptor, so that it no longer refers to any file and may be reused.

Notes:

1. Many functions return a value < 0 to indicate an error you should check this.
2. Make sure you don’t overflow any buffers, and limit the sizes correctly.